CHAPTER

12 Syntactic Parsing

We introduced parsing in Chapter 3 as a combination of recognizing an input string and assigning a structure to it. Syntactic parsing, then, is the task of recognizing a sentence and assigning a syntactic structure to it. This chapter focuses on the kind of structures assigned by context-free grammars of the kind described in Chapter 11. Since they are based on a purely declarative formalism, context-free grammars don't specify *how* the parse tree for a given sentence should be computed. We therefore need to specify algorithms that employ these grammars to efficiently produce correct trees.

Parse trees are directly useful in applications such as **grammar checking** in word-processing systems: a sentence that cannot be parsed may have grammatical errors (or at least be hard to read). More typically, however, parse trees serve as an important intermediate stage of representation for **semantic analysis** (as we show in Chapter 20) and thus play an important role in applications like **question answering** and **information extraction**. For example, to answer the question

What books were written by British women authors before 1800?

we'll need to know that the subject of the sentence was *what books* and that the by-adjunct was *British women authors* to help us figure out that the user wants a list of books (and not a list of authors).

Before presenting any algorithms, we begin by discussing how the ambiguity arises again in this context and the problems it presents. The section that follows then presents the Cocke-Kasami-Younger (CKY) algorithm (Kasami 1965, Younger 1967), the standard dynamic programming approach to syntactic parsing. Recall that we've already seen several applications of dynamic programming algorithms in earlier chapters — Minimum-Edit-Distance, Viterbi, and Forward. Finally, we discuss **partial parsing methods**, for use in situations in which a superficial syntactic analysis of an input may be sufficient.

12.1 Ambiguity

One morning I shot an elephant in my pajamas.
How he got into my pajamas I don't know.
Groucho Marx, *Animal Crackers*, 1930

Ambiguity is perhaps the most serious problem faced by syntactic parsers. Chapter 10 introduced the notions of **part-of-speech ambiguity** and **part-of-speech disambiguation**. Here, we introduce a new kind of ambiguity, called **structural ambiguity**, which arises from many commonly used rules in phrase-structure grammars. To illustrate the issues associated with structural ambiguity, we'll make use of a new toy grammar $\mathscr{L}_1$, shown in Figure 12.1, which consists of the $\mathscr{L}_0$ grammar from the last chapter augmented with a few additional rules.

Structural ambiguity

Grammar	Lexicon
S → *NP VP*	*Det* → *that* | *this* | *the* | *a*
S → *Aux NP VP*	*Noun* → *book* | *flight* | *meal* | *money*
S → *VP*	*Verb* → *book* | *include* | *prefer*
NP → *Pronoun*	*Pronoun* → *I* | *she* | *me*
NP → *Proper-Noun*	*Proper-Noun* → *Houston* | *NWA*
NP → *Det Nominal*	*Aux* → *does*
Nominal → *Noun*	*Preposition* → *from* | *to* | *on* | *near* | *through*
Nominal → *Nominal Noun*	
Nominal → *Nominal PP*	
VP → *Verb*	
VP → *Verb NP*	
VP → *Verb NP PP*	
VP → *Verb PP*	
VP → *VP PP*	
PP → *Preposition NP*	

Figure 12.1 The $\mathscr{L}_1$ miniature English grammar and lexicon.

Structural ambiguity occurs when the grammar can assign more than one parse to a sentence. Groucho Marx's well-known line as Captain Spaulding in *Animal Crackers* is ambiguous because the phrase *in my pajamas* can be part of the *NP* headed by *elephant* or a part of the verb phrase headed by *shot*. Figure 12.2 illustrates these two analyses of Marx's line using rules from $\mathscr{L}_1$.

Structural ambiguity, appropriately enough, comes in many forms. Two common kinds of ambiguity are **attachment ambiguity** and **coordination ambiguity**.

Attachment ambiguity

A sentence has an **attachment ambiguity** if a particular constituent can be attached to the parse tree at more than one place. The Groucho Marx sentence is an example of *PP*-attachment ambiguity. Various kinds of adverbial phrases are also subject to this kind of ambiguity. For instance, in the following example the gerundive-*VP flying to Paris* can be part of a gerundive sentence whose subject is *the Eiffel Tower* or it can be an adjunct modifying the *VP* headed by *saw*:

(12.1) We saw the Eiffel Tower flying to Paris.

Coordination ambiguity

In **coordination ambiguity** different sets of phrases can be conjoined by a conjunction like *and*. For example, the phrase *old men and women* can be bracketed as *[old [men and women]]*, referring to *old men* and *old women*, or as *[old men] and [women]*, in which case it is only the men who are old.

These ambiguities combine in complex ways in real sentences. A program that summarized the news, for example, would need to be able to parse sentences like the following from the Brown corpus:

(12.2) President Kennedy today pushed aside other White House business to devote all his time and attention to working on the Berlin crisis address he will deliver tomorrow night to the American people over nationwide television and radio.

This sentence has a number of ambiguities, although since they are semantically unreasonable, it requires a careful reading to see them. The last noun phrase could be parsed *[nationwide [television and radio]]* or *[[nationwide television] and radio]*. The direct object of *pushed aside* should be *other White House business* but could also be the bizarre phrase *[other White House business to devote all his time and attention to working]* (i.e., a structure like *Kennedy affirmed [his intention to propose*

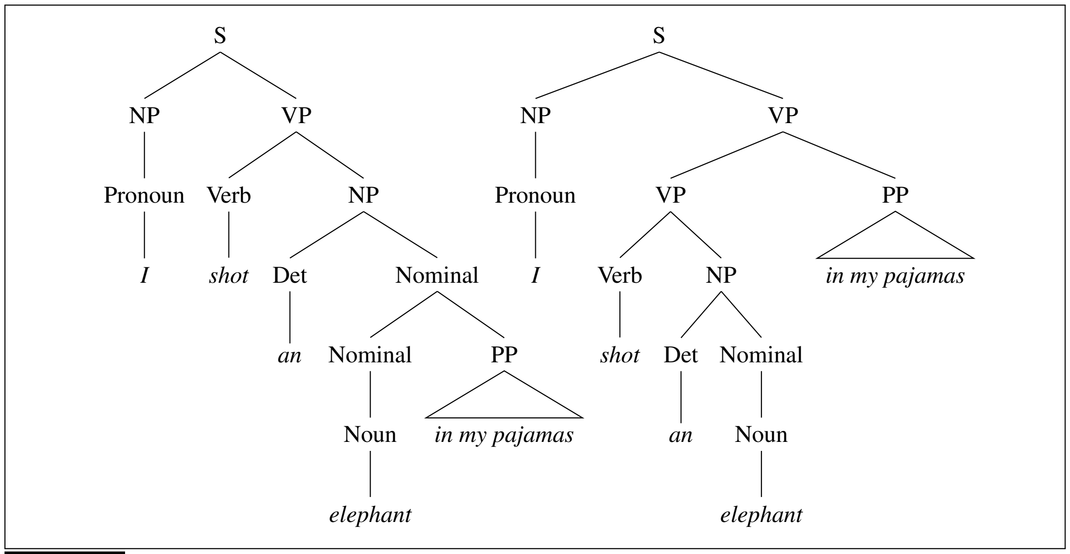

Figure 12.2 Two parse trees for an ambiguous sentence. The parse on the left corresponds to the humorous reading in which the elephant is in the pajamas, the parse on the right corresponds to the reading in which Captain Spaulding did the shooting in his pajamas.

a new budget to address the deficit]). Then the phrase *on the Berlin crisis address he will deliver tomorrow night to the American people* could be an adjunct modifying the verb *pushed*. A *PP* like *over nationwide television and radio* could be attached to any of the higher *VP*s or *NP*s (e.g., it could modify *people* or *night*).

The fact that there are many grammatically correct but semantically unreasonable parses for naturally occurring sentences is an irksome problem that affects all parsers. Ultimately, most natural language processing systems need to be able to choose a single correct parse from the multitude of possible parses through a process of **syntactic disambiguation**. Effective disambiguation algorithms require statistical, semantic, and contextual knowledge sources that vary in how well they can be integrated into parsing algorithms.

Syntactic disambiguation

Fortunately, the CKY algorithm presented in the next section is designed to efficiently handle structural ambiguities of the kind we've been discussing. And as we'll see in Chapter 13, there are straightforward ways to integrate statistical techniques into the basic CKY framework to produce highly accurate parsers.

12.2 CKY Parsing: A Dynamic Programming Approach

The previous section introduced some of the problems associated with ambiguous grammars. Fortunately, **dynamic programming** provides a powerful framework for addressing these problems, just as it did with the Minimum Edit Distance, Viterbi, and Forward algorithms. Recall that dynamic programming approaches systematically fill in tables of solutions to sub-problems. When complete, the tables contain the solution to all the sub-problems needed to solve the problem as a whole. In the case of syntactic parsing, these sub-problems represent parse trees for all the constituents detected in the input.

The dynamic programming advantage arises from the context-free nature of our grammar rules — once a constituent has been discovered in a segment of the input we can record its presence and make it available for use in any subsequent derivation that might require it. This provides both time and storage efficiencies since subtrees can be looked up in a table, not reanalyzed. This section presents the Cocke-Kasami-Younger (CKY) algorithm, the most widely used dynamic-programming based approach to parsing. Related approaches include the **Earley algorithm** (Earley, 1970) and **chart parsing** (Kaplan 1973, Kay 1982).

12.2.1 Conversion to Chomsky Normal Form

We begin our investigation of the CKY algorithm by examining the requirement that grammars used with it must be in Chomsky Normal Form (CNF). Recall from Chapter 11 that grammars in CNF are restricted to rules of the form $A \rightarrow B\ C$ or $A \rightarrow w$. That is, the right-hand side of each rule must expand either to two non-terminals or to a single terminal. Restricting a grammar to CNF does not lead to any loss in expressiveness, since any context-free grammar can be converted into a corresponding CNF grammar that accepts exactly the same set of strings as the original grammar.

Let's start with the process of converting a generic CFG into one represented in CNF. Assuming we're dealing with an ϵ-free grammar, there are three situations we need to address in any generic grammar: rules that mix terminals with non-terminals on the right-hand side, rules that have a single non-terminal on the right-hand side, and rules in which the length of the right-hand side is greater than 2.

The remedy for rules that mix terminals and non-terminals is to simply introduce a new dummy non-terminal that covers only the original terminal. For example, a rule for an infinitive verb phrase such as *INF-VP* $\rightarrow$ *to VP* would be replaced by the two rules *INF-VP* $\rightarrow$ *TO VP* and *TO* $\rightarrow$ *to*.

Unit productions

Rules with a single non-terminal on the right are called **unit productions**. We can eliminate unit productions by rewriting the right-hand side of the original rules with the right-hand side of all the non-unit production rules that they ultimately lead to. More formally, if $A \overset{*}{\Rightarrow} B$ by a chain of one or more unit productions and $B \rightarrow \gamma$ is a non-unit production in our grammar, then we add $A \rightarrow \gamma$ for each such rule in the grammar and discard all the intervening unit productions. As we demonstrate with our toy grammar, this can lead to a substantial *flattening* of the grammar and a consequent promotion of terminals to fairly high levels in the resulting trees.

Rules with right-hand sides longer than 2 are normalized through the introduction of new non-terminals that spread the longer sequences over several new rules. Formally, if we have a rule like

$$A \rightarrow B\ C\ \gamma$$

we replace the leftmost pair of non-terminals with a new non-terminal and introduce a new production result in the following new rules:

$$\begin{aligned} A &\rightarrow X1\ \gamma \\ X1 &\rightarrow B\ C \end{aligned}$$

In the case of longer right-hand sides, we simply iterate this process until the offending rule has been replaced by rules of length 2. The choice of replacing the leftmost pair of non-terminals is purely arbitrary; any systematic scheme that results in binary rules would suffice.

In our current grammar, the rule $S \rightarrow$ *Aux NP VP* would be replaced by the two rules $S \rightarrow$ *X1 VP* and *X1* $\rightarrow$ *Aux NP*.

$\mathscr{L}_1$ **Grammar**	$\mathscr{L}_1$ **in CNF**
S → NP VP	*S → NP VP*
S → Aux NP VP	*S → X1 VP*
	X1 → Aux NP
S → VP	*S → book | include | prefer*
	S → Verb NP
	S → X2 PP
	S → Verb PP
	S → VP PP
NP → Pronoun	*NP → I | she | me*
NP → Proper-Noun	*NP → TWA | Houston*
NP → Det Nominal	*NP → Det Nominal*
Nominal → Noun	*Nominal → book | flight | meal | money*
Nominal → Nominal Noun	*Nominal → Nominal Noun*
Nominal → Nominal PP	*Nominal → Nominal PP*
VP → Verb	*VP → book | include | prefer*
VP → Verb NP	*VP → Verb NP*
VP → Verb NP PP	*VP → X2 PP*
	X2 → Verb NP
VP → Verb PP	*VP → Verb PP*
VP → VP PP	*VP → VP PP*
PP → Preposition NP	*PP → Preposition NP*

Figure 12.3 $\mathscr{L}_1$ Grammar and its conversion to CNF. Note that although they aren't shown here, all the original lexical entries from $\mathscr{L}_1$ carry over unchanged as well.

The entire conversion process can be summarized as follows:

1. Copy all conforming rules to the new grammar unchanged.
2. Convert terminals within rules to dummy non-terminals.
3. Convert unit-productions.
4. Make all rules binary and add them to new grammar.

Figure 12.3 shows the results of applying this entire conversion procedure to the $\mathscr{L}_1$ grammar introduced earlier on page 2. Note that this figure doesn't show the original lexical rules; since these original lexical rules are already in CNF, they all carry over unchanged to the new grammar. Figure 12.3 does, however, show the various places where the process of eliminating unit productions has, in effect, created new lexical rules. For example, all the original verbs have been promoted to both *VP*s and to *S*s in the converted grammar.

12.2.2 CKY Recognition

With our grammar now in CNF, each non-terminal node above the part-of-speech level in a parse tree will have exactly two daughters. A two-dimensional matrix can be used to encode the structure of an entire tree. For a sentence of length n, we will work with the upper-triangular portion of an $(n+1) \times (n+1)$ matrix. Each cell $[i, j]$ in this matrix contains the set of non-terminals that represent all the constituents that span positions i through j of the input. Since our indexing scheme begins with 0, it's natural to think of the indexes as pointing at the gaps between the input words (as in $_0$ *Book* $_1$ *that* $_2$ *flight* $_3$). It follows then that the cell that represents the entire input resides in position $[0, n]$ in the matrix.

Since each non-terminal entry in our table has two daughters in the parse, it follows that for each constituent represented by an entry $[i, j]$, there must be a position in the input, k, where it can be split into two parts such that $i < k < j$. Given such a position k, the first constituent $[i,k]$ must lie to the left of entry $[i, j]$ somewhere along row i, and the second entry $[k, j]$ must lie beneath it, along column j.

To make this more concrete, consider the following example with its completed parse matrix, shown in Fig. 12.4.

(12.3) Book the flight through Houston.

The superdiagonal row in the matrix contains the parts of speech for each input word in the input. The subsequent diagonals above that superdiagonal contain constituents that cover all the spans of increasing length in the input.

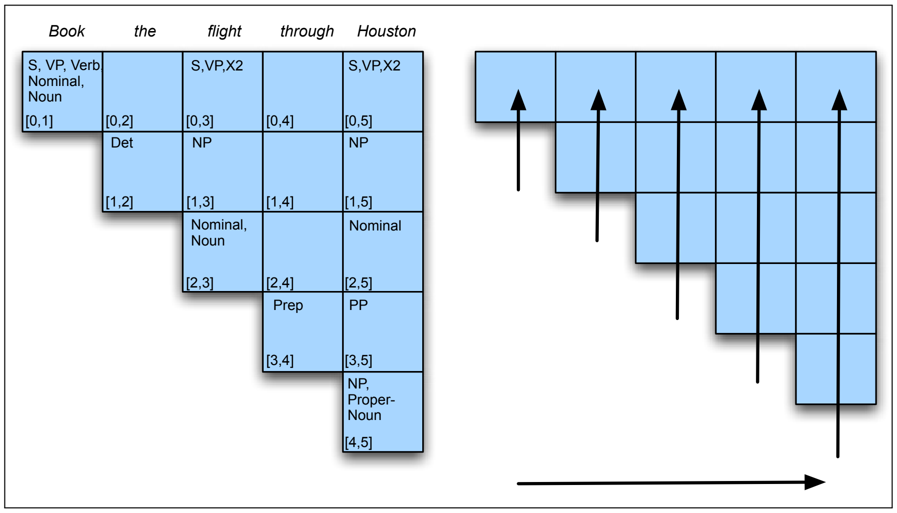

Figure 12.4 Completed parse table for *Book the flight through Houston.*

Given this setup, CKY recognition consists of filling the parse table in the right way. To do this, we'll proceed in a bottom-up fashion so that at the point where we are filling any cell $[i, j]$, the cells containing the parts that could contribute to this entry (i.e., the cells to the left and the cells below) have already been filled. The algorithm given in Fig. 12.5 fills the upper-triangular matrix a column at a time working from left to right, with each column filled from bottom to top, as the right side ofFig. 12.4 illustrates. This scheme guarantees that at each point in time we have all the information we need (to the left, since all the columns to the left have already been filled, and below since we're filling bottom to top). It also mirrors on-line parsing since filling the columns from left to right corresponds to processing each word one at a time.

The outermost loop of the algorithm given in Fig. 12.5 iterates over the columns, and the second loop iterates over the rows, from the bottom up. The purpose of the innermost loop is to range over all the places where a substring spanning i to j in the input might be split in two. As k ranges over the places where the string can be split, the pairs of cells we consider move, in lockstep, to the right along row i and down along column j. Figure 12.6 illustrates the general case of filling cell $[i, j]$. At each such split, the algorithm considers whether the contents of the two cells can be combined in a way that is sanctioned by a rule in the grammar. If such a rule exists, the non-terminal on its left-hand side is entered into the table.

```
function CKY-PARSE(words, grammar) returns table

  for $j \leftarrow$ from 1 to LENGTH(words) do
     for all $\{A \mid A \rightarrow words[j] \in grammar\}$
           $table[j-1, j] \leftarrow table[j-1, j] \cup A$
     for $i \leftarrow$ from $j-2$ downto 0 do
         for $k \leftarrow i+1$ to $j-1$ do
            for all $\{A \mid A \rightarrow BC \in grammar$ and $B \in table[i,k]$ and $C \in table[k,j]\}$
                  $table[i,j] \leftarrow table[i,j] \cup A$
```

Figure 12.5 The CKY algorithm.

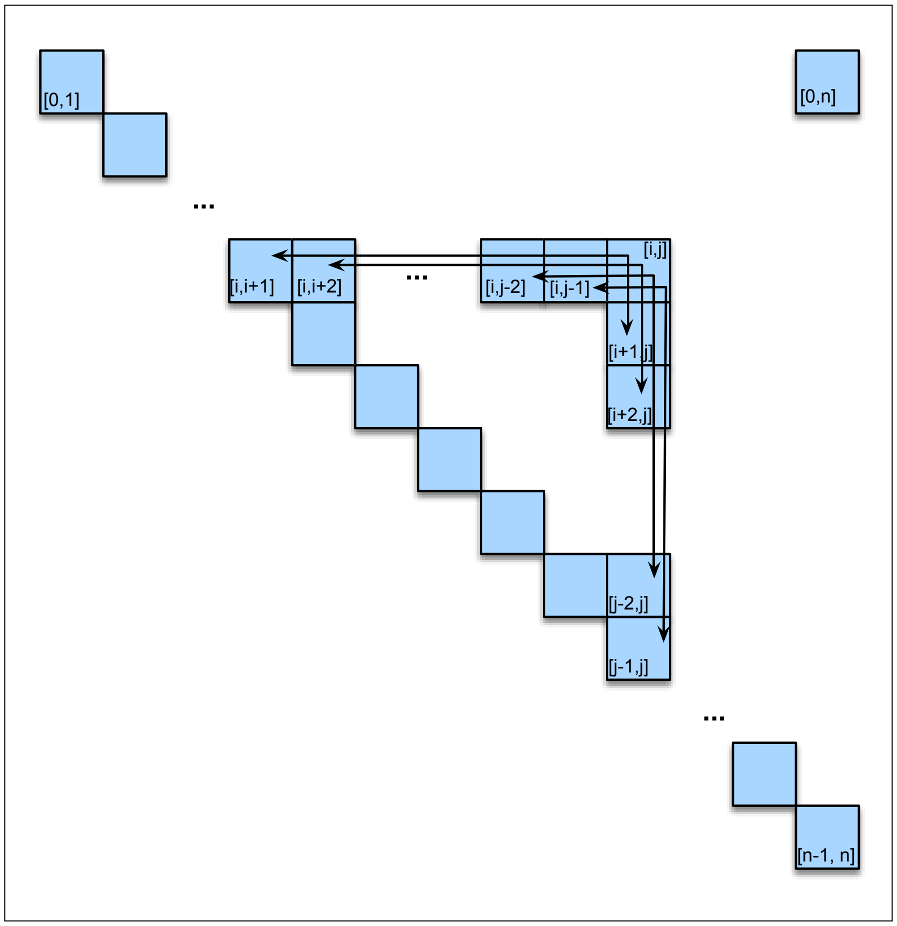

Figure 12.6 All the ways to fill the $[i, j]$th cell in the CKY table.

Figure 12.7 shows how the five cells of column 5 of the table are filled after the word *Houston* is read. The arrows point out the two spans that are being used to add an entry to the table. Note that the action in cell $[0,5]$ indicates the presence of three alternative parses for this input, one where the *PP* modifies the *flight*, one where it modifies the booking, and one that captures the second argument in the original *VP* $\rightarrow$ *Verb NP PP* rule, now captured indirectly with the *VP* $\rightarrow$ *X2 PP* rule.

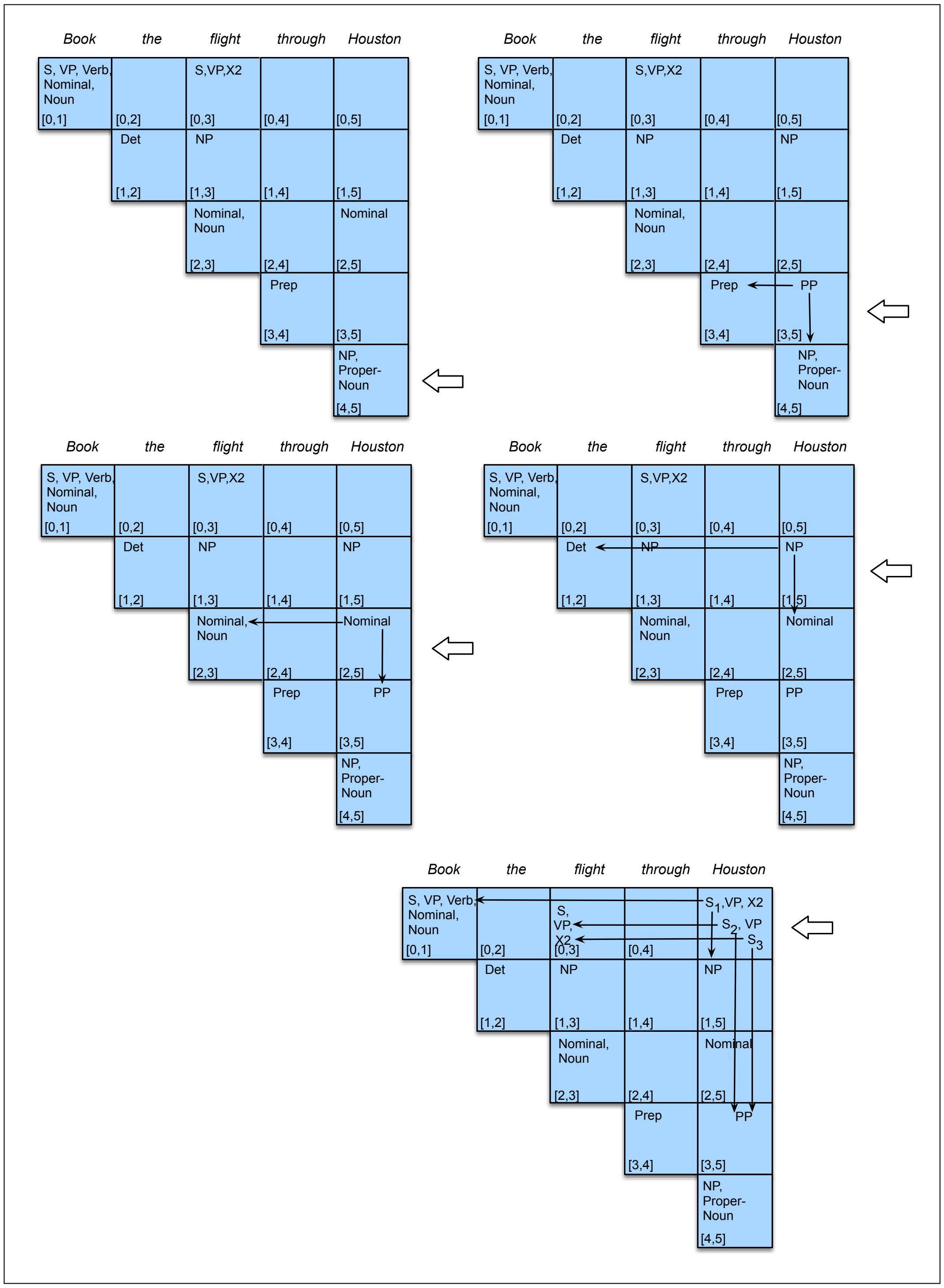

Figure 12.7 Filling the cells of column 5 after reading the word *Houston*.

12.2.3 CKY Parsing

The algorithm given in Fig. 12.5 is a recognizer, not a parser; for it to succeed, it simply has to find an S in cell $[0,n]$. To turn it into a parser capable of returning all possible parses for a given input, we can make two simple changes to the algorithm: the first change is to augment the entries in the table so that each non-terminal is paired with pointers to the table entries from which it was derived (more or less as shown in Fig. 12.7), the second change is to permit multiple versions of the same non-terminal to be entered into the table (again as shown in Fig. 12.7). With these changes, the completed table contains all the possible parses for a given input. Returning an arbitrary single parse consists of choosing an S from cell $[0,n]$ and then recursively retrieving its component constituents from the table.

Of course, returning all the parses for a given input may incur considerable cost since an exponential number of parses may be associated with a given input. In such cases, returning all the parses will have an unavoidable exponential cost. Looking forward to Chapter 13, we can also think about retrieving the best parse for a given input by further augmenting the table to contain the probabilities of each entry. Retrieving the most probable parse consists of running a suitably modified version of the Viterbi algorithm from Chapter 10 over the completed parse table.

12.2.4 CKY in Practice

Finally, we should note that while the restriction to CNF does not pose a problem theoretically, it does pose some non-trivial problems in practice. Obviously, as things stand now, our parser isn't returning trees that are consistent with the grammar given to us by our friendly syntacticians. In addition to making our grammar developers unhappy, the conversion to CNF will complicate any syntax-driven approach to semantic analysis.

One approach to getting around these problems is to keep enough information around to transform our trees back to the original grammar as a post-processing step of the parse. This is trivial in the case of the transformation used for rules with length greater than 2. Simply deleting the new dummy non-terminals and promoting their daughters restores the original tree.

In the case of unit productions, it turns out to be more convenient to alter the basic CKY algorithm to handle them directly than it is to store the information needed to recover the correct trees. Exercise 12.3 asks you to make this change. Many of the probabilistic parsers presented in Chapter 13 use the CKY algorithm altered in just this manner. Another solution is to adopt a more complex dynamic programming solution that simply accepts arbitrary CFGs. The next section presents such an approach.

12.3 Partial Parsing

Partial parse
Shallow parse

Many language processing tasks do not require complex, complete parse trees for all inputs. For these tasks, a **partial parse**, or **shallow parse**, of input sentences may be sufficient. For example, information extraction systems generally do not extract *all* the possible information from a text: they simply identify and classify the segments in a text that are likely to contain valuable information. Similarly, information retrieval systems may index texts according to a subset of the constituents found in them.

There are many different approaches to partial parsing. Some make use of cascades of FSTs, of the kind discussed in Chapter 3, to produce tree-like representations. These approaches typically produce flatter trees than the ones we've been discussing in this chapter and the previous one. This flatness arises from the fact that FST cascade approaches generally defer decisions that may require semantic or contextual factors, such as prepositional phrase attachments, coordination ambiguities, and nominal compound analyses. Nevertheless, the intent is to produce parse trees that link all the major constituents in an input.

Chunking

An alternative style of partial parsing is known as **chunking**. Chunking is the process of identifying and classifying the flat, non-overlapping segments of a sentence that constitute the basic non-recursive phrases corresponding to the major parts-of-speech found in most wide-coverage grammars. This set typically includes noun phrases, verb phrases, adjective phrases, and prepositional phrases; in other words, the phrases that correspond to the content-bearing parts-of-speech. Of course, not all applications require the identification of all of these categories; indeed, the most common chunking task is to simply find all the base noun phrases in a text.

Since chunked texts lack a hierarchical structure, a simple bracketing notation is sufficient to denote the location and the type of the chunks in a given example. The following example illustrates a typical bracketed notation.

(12.4) [$_{NP}$ The morning flight] [$_{PP}$ from] [$_{NP}$ Denver] [$_{VP}$ has arrived.]

This bracketing notation makes clear the two fundamental tasks that are involved in chunking: finding the non-overlapping extents of the chunks and assigning the correct label to the discovered chunks.

Note that in this example all the words are contained in some chunk. This will not be the case in all chunking applications. Many words in any input will often fall outside of any chunk, for example, in systems searching for base *NP*s in their inputs, as in the following:

(12.5) [$_{NP}$ The morning flight] from [$_{NP}$ Denver] has arrived.

The details of what constitutes a syntactic base phrase for any given system varies according to the syntactic theories underlying the system and whether the phrases are being derived from a treebank. Nevertheless, some standard guidelines are followed in most systems. First and foremost, base phrases of a given type do not recursively contain any constituents of the same type. Eliminating this kind of recursion leaves us with the problem of determining the boundaries of the non-recursive phrases. In most approaches, base phrases include the headword of the phrase, along with any pre-head material within the constituent, while crucially excluding any post-head material. Eliminating post-head modifiers from the major categories automatically removes the need to resolve attachment ambiguities. Note that this exclusion does lead to certain oddities, such as *PP*s and *VP*s often consisting solely of their heads. Thus, our earlier example *a flight from Indianapolis to Houston on NWA* is reduced to the following:

(12.6) [$_{NP}$ a flight] [$_{PP}$ from] [$_{NP}$ Indianapolis][$_{PP}$ to][$_{NP}$ Houston][$_{PP}$ on][$_{NP}$ NWA]

12.3.1 Machine Learning-Based Approaches to Chunking

State-of-the-art approaches to chunking use supervised machine learning to *train* a chunker by using annotated data as a training set. As described earlier in Chapter 9, we can view this task as one of **sequence labeling**, where a classifier is trained to label each element of the input sequence. Any of the standard approaches to training classifiers apply to this problem.

The first step in such an approach is to cast the chunking process in a way that is amenable to sequence labeling. A particularly fruitful approach has been to treat chunking as a tagging task similar to part-of-speech tagging (Ramshaw and Marcus, 1995). In this approach, a small tagset simultaneously encodes both the segmentation and the labeling of the chunks in the input. The standard way to do this is called

IOB tagging

IOB tagging and is accomplished by introducing tags to represent the beginning (B) and internal (I) parts of each chunk, as well as those elements of the input that are outside (O) any chunk. Under this scheme, the size of the tagset is $(2n+1)$, where n is the number of categories to be classified. The following example shows the bracketing notation of (12.4) on page 10 reframed as a tagging task:

(12.7)	*The*	*morning*	*flight*	*from*	*Denver*	*has*	*arrived*
	B_NP	I_NP	I_NP	B_PP	B_NP	B_VP	I_VP

The same sentence with only the base-NPs tagged illustrates the role of the O tags.

(12.8)	*The*	*morning*	*flight*	*from*	*Denver*	*has*	*arrived.*
	B_NP	I_NP	I_NP	O	B_NP	O	O

Notice that there is no explicit encoding of the end of a chunk in this scheme; the end of any chunk is implicit in any transition from an I or B to a B or O tag. This encoding reflects the notion that when sequentially labeling words, it is generally easier (at least in English) to detect the beginning of a new chunk than it is to know when a chunk has ended. Not surprisingly, a variety of other tagging schemes represent chunks in subtly different ways, including some that explicitly mark the end of constituents. Tjong Kim Sang and Veenstra (1999) describe three variations on this basic tagging scheme and investigate their performance on a variety of chunking tasks.

Given such a scheme, building a chunker consists of training a classifier to label each word of an input sentence with one of the IOB tags from the tagset. Of course, training requires training data consisting of the phrases of interest delimited and marked with the appropriate category. The direct approach is to annotate a representative corpus. Unfortunately, annotation efforts can be both expensive and time consuming. It turns out that the best place to find such data for chunking is in an existing treebank such as the Penn Treebank described in Chapter 11.

Such treebanks provide a complete parse for each corpus sentence, allowing base syntactic phrases to be extracted from the parse constituents. To find the phrases we're interested in, we just need to know the appropriate non-terminal names in the corpus. Finding chunk boundaries requires finding the head and then including the material to the left of the head, ignoring the text to the right. This is somewhat error-prone since it relies on the accuracy of the head-finding rules described in Chapter 11.

Having extracted a training corpus from a treebank, we must now cast the training data into a form that's useful for training classifiers. In this case, each input can be represented as a set of features extracted from a context window that surrounds the word to be classified. Using a window that extends two words before and two words after the word being classified seems to provide reasonable performance. Features extracted from this window include the words themselves, their parts-of-speech, and the chunk tags of the preceding inputs in the window.

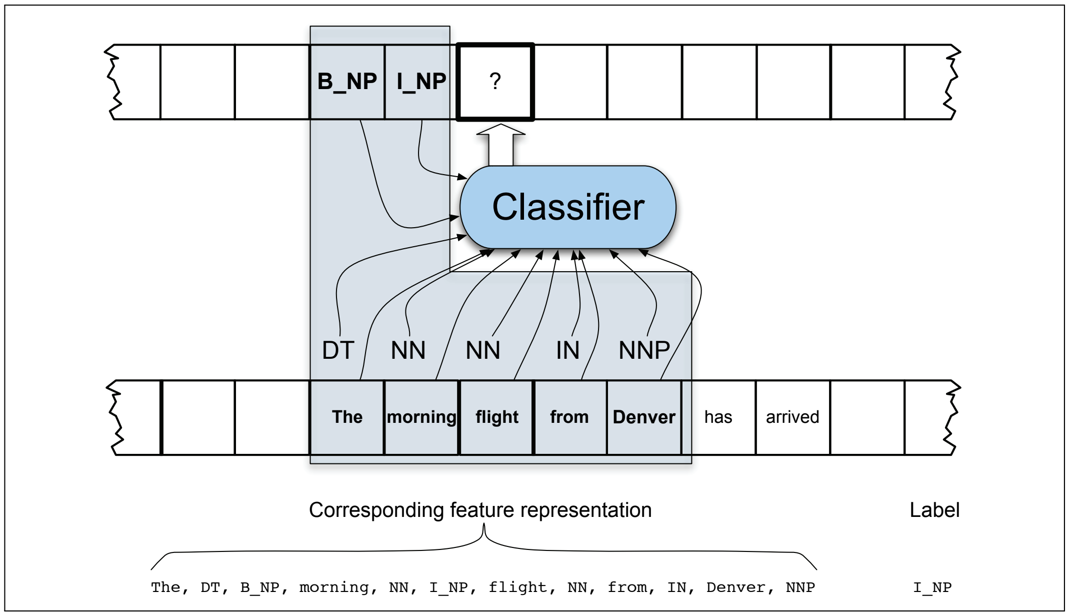
B_NP
I_NP
?
Classifier
DT
NN
NN
IN
NNP
The
morning
flight
from
Denver
has
arrived
Corresponding feature representation
Label
The, DT, B_NP, morning, NN, I_NP, flight, NN, from, IN, Denver, NNP
I_NP

Figure 12.8 A sequential-classifier-based approach to chunking. The chunker slides a context window over the sentence, classifying words as it proceeds. At this point, the classifier is attempting to label *flight*. Features derived from the context typically include the words, part-of-speech tags as well as the previously assigned chunk tags.

Figure 12.8 illustrates this scheme with the example given earlier. During training, the classifier would be provided with a training vector consisting of the values of 13 features; the two words to the left of the decision point, their parts-of-speech and chunk tags, the word to be tagged along with its part-of-speech, the two words that follow along with their parts-of speech, and finally the correct chunk tag, in this case, I_NP. During classification, the classifier is given the same vector without the answer and assigns the most appropriate tag from its tagset.

12.3.2 Chunking-System Evaluations

As with the evaluation of part-of-speech taggers, the evaluation of chunkers proceeds by comparing chunker output with gold-standard answers provided by human annotators. However, unlike part-of-speech tagging, word-by-word accuracy measures are not appropriate. Instead, chunkers are evaluated according to the notions of precision, recall, and the *F*-measure borrowed from the field of information retrieval.

Precision

Precision measures the percentage of system-provided chunks that were correct. Correct here means that both the boundaries of the chunk and the chunk's label are correct. Precision is therefore defined as

$$\textbf{Precision:} = \frac{\text{Number of correct chunks given by system}}{\text{Total number of chunks given by system}}$$

Recall

Recall measures the percentage of chunks actually present in the input that were correctly identified by the system. Recall is defined as

$$\textbf{Recall:} = \frac{\text{Number of correct chunks given by system}}{\text{Total number of actual chunks in the text}}$$

F-measure

The ***F*-measure** (van Rijsbergen, 1975) provides a way to combine these two measures into a single metric. The F-measure is defined as

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P+R}$$

The β parameter differentially weights the importance of recall and precision, based perhaps on the needs of an application. Values of $\beta > 1$ favor recall, while values of $\beta < 1$ favor precision. When $\beta = 1$, precision and recall are equally balanced; this is sometimes called $F_{\beta=1}$ or just F_1:

$$F_1 = \frac{2PR}{P+R} \tag{12.9}$$

F-measure comes from a weighted harmonic mean of precision and recall. The harmonic mean of a set of numbers is the reciprocal of the arithmetic mean of reciprocals:

$$\text{HarmonicMean}(a_1, a_2, a_3, a_4, ..., a_n) = \frac{n}{\frac{1}{a_1}+\frac{1}{a_2}+\frac{1}{a_3}+...+\frac{1}{a_n}} \tag{12.10}$$

and hence F-measure is

$$F = \frac{1}{\alpha\frac{1}{P}+(1-\alpha)\frac{1}{R}} \quad \text{or} \left(\text{with } \beta^2 = \frac{1-\alpha}{\alpha}\right) \quad F = \frac{(\beta^2+1)PR}{\beta^2 P+R} \tag{12.11}$$

Statistical significance results on sequence labeling tasks such as chunking can be computed using matched-pair tests such as McNemar's test, or variants such as the Matched-Pair Sentence Segment Word Error (MAPSSWE) test described on page **??**.

Factors limiting the performance of current systems include part-of-speech tagging accuracy, inconsistencies in the training data introduced by the process of extracting chunks from parse trees, and difficulty resolving ambiguities involving conjunctions. Consider the following examples that involve pre-nominal modifiers and conjunctions.

(12.12) [$_{NP}$ Late arrivals and departures] are commonplace during winter.

(12.13) [$_{NP}$ Late arrivals] and [$_{NP}$ cancellations] are commonplace during winter.

In the first example, *late* is shared by both *arrivals* and *departures*, yielding a single long base-NP. In the second example, *late* is not shared and modifies *arrivals* alone, thus yielding two base-NPs. Distinguishing these two situations, and others like them, requires access to semantic and context information unavailable to current chunkers.

12.4 Summary

The two major ideas introduced in this chapter are those of **parsing** and **partial parsing**. Here's a summary of the main points we covered about these ideas:

- **Structural ambiguity** is a significant problem for parsers. Common sources of structural ambiguity include **PP-attachment**, **coordination ambiguity**, and **noun-phrase bracketing ambiguity**.

- **Dynamic programming** parsing algorithms, such as **CKY**, use a table of partial parses to efficiently parse ambiguous sentences.
- **CKY** restricts the form of the grammar to Chomsky normal form (CNF).
- Many practical problems, including **information extraction** problems, can be solved without full parsing.
- **Partial parsing** and **chunking** are methods for identifying shallow syntactic constituents in a text.
- State-of-the-art methods for partial parsing use **supervised machine learning** techniques.

Bibliographical and Historical Notes

Writing about the history of compilers, Knuth notes:

> In this field there has been an unusual amount of parallel discovery of the same technique by people working independently.

Well, perhaps not unusual, if multiple discovery is the norm (see page **??**). But there has certainly been enough parallel publication that this history errs on the side of succinctness in giving only a characteristic early mention of each algorithm; the interested reader should see Aho and Ullman (1972).

Bottom-up parsing seems to have been first described by Yngve (1955), who gave a breadth-first, bottom-up parsing algorithm as part of an illustration of a machine translation procedure. Top-down approaches to parsing and translation were described (presumably independently) by at least Glennie (1960), Irons (1961), and Kuno and Oettinger (1963). Dynamic programming parsing, once again, has a history of independent discovery. According to Martin Kay (personal communication), a dynamic programming parser containing the roots of the CKY algorithm was first implemented by John Cocke in 1960. Later work extended and formalized the algorithm, as well as proving its time complexity (Kay 1967,Younger 1967,Kasami 1965).
WFST
The related **well-formed substring table** (**WFST**) seems to have been independently proposed by Kuno (1965) as a data structure that stores the results of all previous computations in the course of the parse. Based on a generalization of Cocke's work, a similar data structure had been independently described in Kay 1967, Kay 1973. The top-down application of dynamic programming to parsing was described in Earley's Ph.D. dissertation (Earley 1968, Earley 1970). Sheil (1976) showed the equivalence of the WFST and the Earley algorithm. Norvig (1991) shows that the efficiency offered by dynamic programming can be captured in any language with a *memoization* function (such as in LISP) simply by wrapping the *memoization* operation around a simple top-down parser.

While parsing via cascades of finite-state automata had been common in the early history of parsing (Harris, 1962), the focus shifted to full CFG parsing quite soon afterward. Church (1980) argued for a return to finite-state grammars as a processing model for natural language understanding; other early finite-state parsing models include Ejerhed (1988). Abney (1991) argued for the important practical role of shallow parsing. Much recent work on shallow parsing applies machine learning to the task of learning the patterns; see, for example, Ramshaw and Marcus (1995), Argamon et al. (1998), Munoz et al. (1999).

The classic reference for parsing algorithms is Aho and Ullman (1972); although the focus of that book is on computer languages, most of the algorithms have been applied to natural language. A good programming languages textbook such as Aho et al. (1986) is also useful.

Exercises

12.1 Implement the algorithm to convert arbitrary context-free grammars to CNF. Apply your program to the $\mathscr{L}_1$ grammar.

12.2 Implement the CKY algorithm and test it with your converted $\mathscr{L}_1$ grammar.

12.3 Rewrite the CKY algorithm given in Fig. 12.5 on page 7 so that it can accept grammars that contain unit productions.

12.4 Discuss the relative advantages and disadvantages of partial versus full parsing.

12.5 Discuss how to augment a parser to deal with input that may be incorrect, for example, containing spelling errors or mistakes arising from automatic speech recognition.

Abney, S. P. (1991). Parsing by chunks. In Berwick, R. C., Abney, S. P., and Tenny, C. (Eds.), *Principle-Based Parsing: Computation and Psycholinguistics*, pp. 257–278. Kluwer.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

Aho, A. V. and Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling*, Vol. 1. Prentice Hall.

Argamon, S., Dagan, I., and Krymolowski, Y. (1998). A memory-based approach to learning shallow natural language patterns. In *COLING/ACL-98*, Montreal, pp. 67–73.

Church, K. W. (1980). *On Memory Limitations in Natural Language Processing* Master's thesis, MIT. Distributed by the Indiana University Linguistics Club.

Earley, J. (1968). *An Efficient Context-Free Parsing Algorithm*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, *6*(8), 451–455. Reprinted in Grosz et al. (1986).

Ejerhed, E. I. (1988). Finding clauses in unrestricted text by finitary and stochastic methods. In *ANLP 1988*, pp. 219–227.

Glennie, A. (1960). On the syntax machine and the construction of a universal compiler. Tech. rep. No. 2, Contr. NR 049-141, Carnegie Mellon University (at the time Carnegie Institute of Technology), Pittsburgh, PA.

Harris, Z. S. (1962). *String Analysis of Sentence Structure*. Mouton, The Hague.

Irons, E. T. (1961). A syntax directed compiler for ALGOL 60. *Communications of the ACM*, *4*, 51–55.

Kaplan, R. M. (1973). A general syntactic processor. In Rustin, R. (Ed.), *Natural Language Processing*, pp. 193–241. Algorithmics Press.

Kasami, T. (1965). An efficient recognition and syntax analysis algorithm for context-free languages. Tech. rep. AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

Kay, M. (1967). Experiments with a powerful parser. In *Proc. 2eme Conference Internationale sur le Traitement Automatique des Langues*, Grenoble.

Kay, M. (1973). The MIND system. In Rustin, R. (Ed.), *Natural Language Processing*, pp. 155–188. Algorithmics Press.

Kay, M. (1982). Algorithm schemata and data structures in syntactic processing. In Allén, S. (Ed.), *Text Processing: Text Analysis and Generation, Text Typology and Attribution*, pp. 327–358. Almqvist and Wiksell, Stockholm.

Kuno, S. (1965). The predictive analyzer and a path elimination technique. *Communications of the ACM*, *8*(7), 453–462.

Kuno, S. and Oettinger, A. G. (1963). Multiple-path syntactic analyzer. In Popplewell, C. M. (Ed.), *Information Processing 1962: Proceedings of the IFIP Congress 1962*, Munich, pp. 306–312. North-Holland. Reprinted in Grosz et al. (1986).

Munoz, M., Punyakanok, V., Roth, D., and Zimak, D. (1999). A learning approach to shallow parsing. In *EMNLP/VLC-99*, College Park, MD, pp. 168–178.

Norvig, P. (1991). Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, *17*(1), 91–98.

Ramshaw, L. A. and Marcus, M. P. (1995). Text chunking using transformation-based learning. In *Proceedings of the 3rd Annual Workshop on Very Large Corpora*, pp. 82–94.

Sheil, B. A. (1976). Observations on context free parsing. *SMIL: Statistical Methods in Linguistics*, *1*, 71–109.

Tjong Kim Sang, E. F. and Veenstra, J. (1999). Representing text chunks. In *EACL-99*, pp. 173–179.

van Rijsbergen, C. J. (1975). *Information Retrieval*. Butterworths.

Yngve, V. H. (1955). Syntax and the problem of multiple meaning. In Locke, W. N. and Booth, A. D. (Eds.), *Machine Translation of Languages*, pp. 208–226. MIT Press.

Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control*, *10*, 189–208.